7. Financial Forecasting

1. Understand Recursive Algorithms

Recursion:

- **Concept:** Recursion is a programming technique where a function calls itself in order to solve a problem. The recursive function usually has a base case that stops the recursion and one or more recursive cases that reduce the problem into smaller subproblems.
- **Benefits:**
    - Simplifies code for problems that have a natural recursive structure, such as tree traversal, factorial calculation, and the Fibonacci sequence.
    - Often results in clearer and more readable code compared to iterative solutions for certain problems.

Example: Factorial Calculation:

- **Recursive Definition:**
    - factorial(n) = n * factorial(n - 1) for n > 1
    - factorial(1) = 1 (base case)

Example in Java:

java

Copy code

```
public int factorial(int n) {
    if (n == 1) {
        return 1; // base case
    } else {
        return n * factorial(n - 1); // recursive case
    }
}
```

2. Setup

Financial Forecasting Tool:

- **Scenario:** Predict future values based on past growth rates using a recursive approach.

Assumptions:

- We have historical data of past values.
- We will use a simple growth rate to predict future values.

Example:

- If the past value is PPP and the growth rate is ggg, the future value FFF after nnn periods can be calculated as:
    - F=P×(1+g)n F = P \times (1 + g)^n F=P×(1+g)n

3. Implementation

Recursive Method to Calculate Future Value:

```
package FinancialForecasting;


/**
 *
 * @author Aishwarya
 */
import java.util.Scanner;

public class FinancialForecasting {


    public static double calculateFutureValue(double presentValue, double growthRate, int years) {


        if (years == 0) {
            return presentValue;
        }


        return calculateFutureValue(presentValue * (1 + growthRate), growthRate, years - 1);
```

```
    }

public static void main(String[] args) { Scanner
scanner = new Scanner(System.in);

        System.out.print("Enter the present value: ");
        double presentValue = scanner.nextDouble();

        System.out.print("Enter the growth rate (as a decimal, e.g., 0.05 for 5%): ");
        double growthRate = scanner.nextDouble();

        System.out.print("Enter the number of years: ");
        int years = scanner.nextInt();

        double futureValue = calculateFutureValue(presentValue, growthRate, years);

        System.out.printf("The future value after %d years is: %.2f%n", years,
    futureValue);

        scanner.close();
    }
    public static double calculateFutureValueIterative(double presentValue, double
    growthRate, int years) {
    double futureValue = presentValue;
    for (int i = 0; i < years; i++) {
        futureValue *= (1 + growthRate);
    }
    return futureValue;
}
}
```

4. Analysis

Time Complexity:

- The time complexity of the recursive algorithm is O(n)O(n)O(n) because it makes nnn recursive calls, where nnn is the number of periods.

Optimization to Avoid Excessive Computation:

- **Memoization:** Store the results of previous computations to avoid redundant calculations. This technique transforms a recursive algorithm with overlapping subproblems into a more efficient algorithm.
- **Iterative Approach:** An iterative solution can be used to eliminate the overhead of recursive function calls, resulting in a more efficient algorithm with the same time complexity.

Optimized Recursive Method with Memoization:

```
import java.util.HashMap;
import java.util.Map;


public class FinancialForecasting {

    private Map<Integer, Double> memo = new HashMap<>();
    public double calculateFutureValue(double presentValue, double growthRate, int periods) {
        if (periods == 0) {
            return presentValue;
        }
        if (memo.containsKey(periods)) {
            return memo.get(periods);
        }
        double futureValue = calculateFutureValue(presentValue * (1 + growthRate), growthRate, periods - 1);
        memo.put(periods, futureValue);
        return futureValue;
    }
```

```
    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();
        double presentValue = 1000.0; // Example present value
        double growthRate = 0.05; // Example growth rate (5%)
        int periods = 10; // Example number of periods

        double futureValue = forecasting.calculateFutureValue(presentValue,
growthRate, periods);
        System.out.println("Future Value after " + periods + " periods: " + futureValue);
    }
}
```

Iterative Method:

```
public class FinancialForecasting {
    public double calculateFutureValueIteratively(double presentValue, double
growthRate, int periods) {
        double futureValue = presentValue;
        for (int i = 0; i < periods; i++) {
            futureValue *= (1 + growthRate);
        }
        return futureValue;
    }

    public static void main(String[] args) {
        FinancialForecasting forecasting = new FinancialForecasting();
        double presentValue = 1000.0; // Example present value
        double growthRate = 0.05; // Example growth rate (5%)
        int periods = 10; // Example number of periods
```

```
        double futureValue = forecasting.calculateFutureValueIteratively(presentValue, growthRate, periods);
        System.out.println("Future Value after " + periods + " periods: " + futureValue);
    }
}
```

Conclusion

- **Recursive Approach:**
  - Easy to understand and implement for problems with a natural recursive structure.
  - Can be optimized using memoization to avoid excessive computations.
- **Iterative Approach:**
  - More efficient in terms of space and time for this specific problem.
  - Eliminates the overhead of recursive function calls.

For financial forecasting, where the number of periods might be large, the iterative approach or optimized recursive approach with memoization is recommended to ensure efficiency and avoid stack overflow issues.